Multiple issue and VLIW processors

To improve performance beyond what hardware-based speculation can provide, we would like to decrease the CPI to less than one, but this cannot happen unless we issue more than one instruction per clock cycle.

The goal of *multiple-issue processors* is to allow multiple instructions to issue in a clock cycle.

Multiple-issue processors come in three main varieties:

1. Statically scheduled superscalar processors
2. *Very Long Instruction Word* (VLIW) processors
3. Dynamically scheduled superscalar processors

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one, large instruction or as a fixed instruction packet, with the parallelism among the instructions explicitly indicated by the instruction.

VLIW processors are inherently statically scheduled by the compiler.

The VLIW approach

Rather than attempting to issue multiple independent instructions to its functional units, VLIW processors package operations into a single very long instruction.

The advantage of a VLIW increases as the maximum issue rate grows; however, the growth in overhead when scheduling many instructions is a major limiting factor for this type of processor.

In order to keep functional units busy, there must be enough parallelism in a single instruction packet to fill the available operation slots. This parallelism is uncovered by loop unrolling, which is then scheduled statically. If the unrolling generates straight-line code, then *local scheduling* techniques, which operate on a single block, can be used, while if finding the parallelism among the instructions requires scheduling code across branches, a more complex *global scheduling* algorithm must be used.

Issues

For the original VLIW model, there were both technical and logistical problems that made the approach less efficient.

Increase in code size

Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops, thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding.

To combat this size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded.

Lockstep operation

Early VLIW operated in lockstep: there was no hazard-detection hardware at all. This structure dictated that a stall in any functional unit must cause **the entire processor to stall**, since all functional units must be kept synchronised.

For this reason, caches needed to be blocking and to cause all the functional units to stall.

As the issue rate and number of memory references becomes larger, this synchronisation restriction becomes unacceptable.

In more recent processors, the functional units operate more independently and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronised execution once instructions are issued.

Code compatibility

Binary code compatibility has also been a major logistical problem for VLIW processors. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between implementations more difficult than it is for a superscalar design.

The superscalar, multiple-issue approach

So far, we have seen how the individual mechanisms of dynamic scheduling, multiple issue and speculation work. Now, we can put all three together to obtain a very efficient architecture which is similar to the one found in today's processors.

For simplicity, we consider an issue rate of two instructions per clock, but the concepts are exactly the same for higher-issue-rate processors.

Let's assume we want to extend Tomasulo's algorithm to support multiple-issue superscalar pipelines with separate integer, load and store and floating point units, each of which can initiate an operation on every clock.

To gain the full advantage of dynamic scheduling, we will allow the pipeline to issue any combination of two instructions in a clock. Furthermore, we incorporate speculative execution.

Issuing multiple instructions per clock in a dynamically scheduled processors—with or without speculation—is a very complex matter, considering all the dependencies that could arise.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor. Both rely on the observation that the key is assigning a reservation station and updating the pipeline control tables.

One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle. However, this approach is not very scalable: it cannot be easily extended to more than two instructions per clock.

A second alternative is to build the logic necessary to handle two or more instructions per clock in a dynamically scheduled processor, including any possible dependencies between the instructions.

Modern superscalar processors that issue four or more instructions per clock may include both approaches.

A key observation is that we **cannot simply pipeline away the problem**. By making instructions issues take multiple clocks because new instructions are issuing every clock cycle, we must be able to assign the reservation station and to update the pipeline tables so that a dependent instruction issuing on the next clock can use the updated information. This issue step is one of the **most fundamental bottlenecks in dynamically scheduled superscalars**.

Multiple-issue strategy

We can generalise the basic strategy for updating the issue logic and the reservation tables in a dynamically scheduled superscalar processor with up to n issues per clock as follows:

1. Assign a reservation station and a ROB for **every instruction that might be issued in the next issue bundle**. This assignment can be done before the instruction types are known by simply preallocating the reorder buffer entries sequentially to the instructions in the packet using n available ROB entries and by ensuring that enough reservation stations are available to issue the whole bundle. Should sufficient reservation stations not be available, the bundle is broken and only a subset of the instructions is issued. The remainder of the instructions can be placed in the next bundle for potential issue.
2. Analyse all the dependences among the instructions in the issue bundle.
3. If an instruction in the bundle depends on an earlier instruction in the same bundle, use the assigned ROB number to update the reservation table for the dependent instruction. Otherwise, use the existing reservation table and ROB information to update the reservation table entries for the issuing instruction.

What makes the above very complicated is that it is all done in parallel in a single clock cycle.

At the back-end of the pipeline, we must be able to complete and commit multiple instructions per clock. These steps are somewhat easier than the issue problems, since multiple instructions that can actually commit in the same clock cycle must have already dealt with and resolved any dependences.